Assessing the Performance of the SRR Loop Scheduler with Irregular Workloads

Pedro Henrique Penna, Eduardo C Inacio, Márcio Castro, Patrícia Plentz, Henrique Cota de Freitas, François Broquedis, Jean-François Méhaut

Assessing the Performance of the SRR Loop Scheduler with Irregular Workloads

Pedro H. Penna[1], Eduardo C. Inacio[1], Márcio Castro[1], Patrícia Plentz[1], Henrique C. Freitas[2], François Broquedis[3], and Jean-François Méhaut[3]

[1] Federal University of Santa Catarina, Florianópolis, Brazil
[2] Pontifical Catholic University of Minas Gerais, Belo Horizonte, Brazil
[3] University of Grenoble Alpes, Grenoble, France

Abstract
The input workload of an irregular application must be evenly distributed among its threads to enable cutting-edge performance. To address this need in OpenMP, several loop scheduling strategies were proposed. While having this ever-increasing number of strategies at disposal is helpful, it has become a non-trivial task to select the best one for a particular application. Nevertheless, this challenge becomes easier to be tackled when existing scheduling strategies are extensively evaluated. Therefore, in this paper, we present a performance and scalability evaluation of the recently-proposed loop scheduling strategy named Smart Round-Robin (SRR). To deliver a comprehensive analysis, we coupled a synthetic kernel benchmarking technique with several rigorous statistical tools, and considered OpenMP's Static and Dynamic loop schedulers as our baselines. Our results unveiled that SRR performs better on irregular applications with symmetric workloads and coarse-grained parallelization, achieving up to 1.9x and 1.5x speedup over OpenMP's Static and Dynamic schedulers, respectively.

Keywords: Irregular Workloads, Loop Scheduling, Performance Evaluation, Kernel Benchmarking

1 Introduction

In High Performance Computing (HPC), parallel applications can be classified as either *regular* or *irregular*. In the former group, the time needed to solve a given problem is strictly related to the size of the input data. A naive implementation of the matrix multiplication algorithm is a typical example of application that belongs to this group, in which the number of operations is constantly proportional to the products of the rows of a matrix by the columns of another matrix, regardless of the actual numbers involved in the computation. On the other hand, in the latter group, the contents of the input data also impact significantly on their execution times [6]. For instance, in a data clustering application that uses a Minimum Spanning Tree algorithm, the computation time depends on how the data is distributed in the Clustering Euclidean space.

In the case of regular parallel applications, the input workload can be naturally broken up into homogeneous tasks, so that no thread is greatly overloaded with computations. In the case of irregular applications, in contrast, the workload may not be so easily divided, and thus load imbalance may arise among the threads. This may result in considerable performance and scalability issues, since the overall performance of the application would be bounded by the performance of the most overloaded thread.

Indeed, evenly distributing the input workload of an irregular application to its threads is a well-known NP-Hard problem named the Load Balancing Problem [9], and it is a recurring subject of research in HPC [3, 4, 8]. For instance, in OpenMP, an industry and academia standard Application Programming Interface (API) for parallel programming on shared-memory architectures [2], this problem arises when scheduling iterations in parallel loops. In this context, the problem is referenced as the Loop Scheduling Problem and comes down into assigning loop iterations to threads, so as to evenly distribute the overall load between them.

To address this new problem, several loop scheduling strategies were proposed to cover a great variety of scenarios. They aim at mitigating the load imbalance between iterations by smartly assigning them to threads [3, 5, 8]. While having this ever-increasing number of strategies at disposal is indeed helpful, it has become a complex task to actually select the best one for a particular application [12, 10].

In this paper we argue that this challenge becomes easier to be tackled when existing loop scheduling strategies are extensively evaluated. Therefore, the main goal of this work is to evaluate the performance bounds and scaling capabilities of the recently-proposed loop scheduling strategy named Smart Round-Robin (SRR) [8]. This strategy showed promising results for irregular applications, but has not yet been thoroughly assessed. More precisely, we present the following contributions in this paper: (i) a detailed and comprehensive performance analysis of SRR through a full factorial experimental design, which considers six performance factors that impact on the workload of an irregular application, coupled with rigorous statistical tools to deliver a statistically significant assessment; (ii) an analysis of the weak and strong scalability potentials of this emerging strategy; and (iii) a throughout comparison evaluation of SRR against the OpenMP's Static and Dynamic loop scheduling strategies.

The remainder of this work is organized as follows. In Section 2, we present a background of the loop scheduling strategies considered in this paper. In Section 3, we discuss related works and our contributions to the state-of-the-art. In Section 4 discusses our evaluation methodology. In Section 5, we discuss our experimental results. In Section 6, we present the main conclusions of this work and its future perspectives.

2 Loop Scheduling Strategies in OpenMP

OpenMP is shipped with three loop scheduling strategies: Static, Dynamic and Guided [2]. The first one divides the iterations of a parallel loop into user-defined equal-size chunks, which are then statically assigned to threads in a round-robin fashion. This strategy introduces minimum overhead in the application runtime, since scheduling is performed statically. It is more suitable for regular applications, because it schedules chunks regardless of their load.

In contrast, the Dynamic strategy uses an internal work queue to dynamically assign chunk-sized blocks of loop iterations to threads. The scheduling is performed on-demand at cost of some performance overhead, and the chunk size may be fine-tuned to achieve load balancing [1]. Therefore, this strategy is recommended to parallel applications that feature irregular behavior. Finally, the Guided loop scheduling strategy works similar to the Dynamic one, but the chunk size starts off large and decreases with the course of time. This allows for a better tradeoff

between the synchronization overhead and load balancing because: (i) threads synchronize less frequently to access the internal work queue at the beginning due to the large chunk sizes; and (ii) as the execution approaches the end, the chunk size becomes small enough to guarantee a better load balance.

SRR is a workload-aware loop scheduling strategy recently-proposed to address irregular applications that may have their input workload somehow estimated [8]. Unlike the other strategies shipped with OpenMP, this one considers some information about the input workload of the application to better balance the load among the threads. SRR was implemented in *libgomp*, the GCC's OpenMP runtime library, and it is publicly available. The main idea behind SRR is to assign pairs of chunk-sized loop iterations to threads in a round-robin fashion, so that, in the end, each thread is assigned to a near average workload. For this, pairs are formed up by chunk-sized blocks of iterations that have not yet been assigned to some thread and have the lowest and highest loads. A complete description of SRR can be found in [8].

3 Related Work

To cover a great variety of scenarios, several loop scheduling strategies were proposed. Targeting memory-bound irregular applications running on large-scale NUMA platforms, Durand *et al.* introduced a new loop scheduler called *Adaptative* [4]. This strategy uses a work-stealling algorithm to dynamically adapt the chunk granularity in parallel loops, and thus better exploit memory affinity. Their experimental results unveiled that *Adaptative* overpasses the OpenMP's Dynamic scheduler in irregular applications, while delivering equi-performance to OpenMP's Static on regular applications. Other memory-affinity schedulers are discussed in [3, 7].

In contrast to scheduling strategies that focus on exploiting runtime information, Thoman *et al.* introduced an alternative hybrid approach that uses compiling time information in additon [13]. They implemented their loop scheduler in the *Insieme Compiler* and runtime system, and contrasted its performance with OpenMP's default loop schedulers. Their results suggested that a hybrid scheduler may even superior strategies than the scheduling strategies available in OpenMP. In [5] compiling information is also considered in loop scheduling.

With this ever-increasing number of loop scheduling strategies, the task of actually selecting the best one for a particular scenario has become non-trivial. To address this challenge, Sukhija *et al.* proposed a prediction algorithm based on Machine Learning that selects the most robust loop scheduling strategy for a target application/platform [12]. Based on their results, they concluded that their approach selects the most robust loop scheduling strategy, given a user-specified tolerance. Targeting a similar goal, Srivastava *et al.* proposed a strategy based on Artificial Neural Networks (ANNs) to predict the performance of dynamic loop scheduling strategies on heterogeneous platforms [10]. To train the ANN, they used results obtained with a synthetic kernel benchmark running on synthetically-generated input workloads based the Gamma, Gaussian and Exponential Probability Density Functions (PDFs). Their results unveiled that the proposed strategy is able to predict the performance of a dynamic scheduling strategy, and thus can guide the selection of the best strategy on heterogeneous platforms.

What concerns the efforts for assessing scheduling strategies, Srivastava *et al.* proposed a methodology for evaluating the performance of dynamic loop schedulers [11]. Their methodology relies on the simulation and synthetic kernel benchmarking techniques, and considers Gaussian-generated irregular workloads. In the end, they concluded that their methodology may be applied to evaluate the performance of loop scheduling strategies. Another work that proposes a similar evaluation methodology is discussed in [1].

Our work differs from the previous ones in three main points. First, unlike those works that

proposed loop scheduling strategies [4, 3, 7, 13, 5, 8], in this work we focus on the actual performance and scalability evaluation of a recently-proposed strategy, the SRR scheduler. Second, in contrast to the related works that employed simulation and synthetic kernel benchmarking for evaluating loop scheduling strategies [11, 1, 12, 10], in this work we couple the latter technique with several rigorous statistical tools in addition, to deliver a comprehensive and statistically significant analysis. Finally, we carry out weak and strong scaling experiments to unveil the scalability potentials of SRR.

4 Evaluation Methodology

This section describes the performance factors considered in the evaluation. Then, it details the experimental design method that we followed in our experiments.

4.1 Performance Factors

We considered the following six performance factors to assess the performance of the SRR loop scheduling strategy: (i) the input workload PDF; (ii) the input workload PDF's kurtosis; (iii) the loop iteration shuffling; (iv) the complexity of the application kernel; (v) the number of chunks of loop iterations; and (vi) the number of threads. The last three factors model the irregular application, whereas the other ones model its input workload.

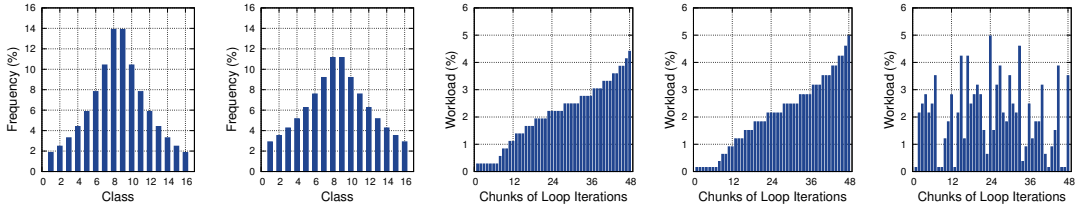
Workload PDF (PDF). It models the frequency of light, medium and heavy load chunks of loop iterations in the input workload. Chunks of loop iterations belonging to the same *class* have the same load. The more skewed the PDF is, the stronger is the irregularity in the input workload and more difficult is to achieve load balancing.

Workload PDF's Kurtosis (Kurtosis). It models how strong is the frequency change in the PDF of the input workload. The stronger this factor is, the stronger is the irregularity in the input workload and more difficult is to evenly distribute the workload.

Loop Chunk Shuffling (Shuffling). It states how chunks of iterations are shuffled in the input workload. In blind loop scheduling strategies, *i.e.,* those that do not consider any information about the input workload, this performance factor may greatly impact on their load balancing capability.

Complexity of the Application Kernel (Kernel). It models the runtime complexity of the irregular application. The more complex is the kernel the stronger is the impact of the input workload on runtime.

Number of Chunks of Loop Iterations. It models the granularity level of parallelization in the application. The higher is the number of chunks the easier is to balance the overall input



(a) PDF. (b) Kurtosis. (c) Input workload. (d) Kernel. (e) Shuffling.

Figure 1: Impact of performance factors in the application's workload.

workload, but the higher is the synchronization and communication overheads.

Number of Threads. It states the number of working threads in the parallel application. The more threads, the shorter should be the time-to-solution of the application.

Figure 1 illustrates the impact of some of these performance factors in the workload of the application. In this example, the frequency of each chunk load class was generated according to the Gaussian PDF (Figure 1a). Chunk load class frequencies are fine adjusted by the PDF kurtosis (Figure 1b). Each chunk load class is assigned to a different load, resulting in the input workload of the application (Figure 1c). The complexity of the application kernel strengthens the load imbalance between chunks of loop iterations (Figure 1d). Finally, loop chunk shuffling models how chunks of loop iterations are actually disposed in a for loop (Figure 1e).

4.2 Experimental Design

To assess the performance bounds and scaling capabilities of SRR, we carried out three experiments considering the performance factors presented earlier, and using the synthetic kernel proposed in [8]. We considered the following levels for performance factors: *PDF* = { Beta, Gamma Gaussian, Uniform }; *Kurtosis* = { 0.750, 0.775, 0.800, 0.825 0.850, 0.875, 0.900 }; *Shuffling* = { 1, 307, 769, 967 }; *Kernel* = { Linear, Logarithmic }; *Chunks* = { 4, 8, 12, 16, 20, 24 28, 32, 36, 40, 44, 48 }; and *Threads* = { 2, 4, 6, 8, 10, 12 14, 16, 18, 20, 24 }.

In the first experiment, we intend to compare the performance of SRR with the OpenMP's Static and Dynamic strategies. To do so, we adopt a full factorial experimental design to deliver a comprehensive and statistically significant analysis. We considered the following four performance factors, resulting in 224 possible scenarios for each of the three loop scheduling strategies: PDF, Kurtosis, Shuffling and Kernel. For this experiment, we set the number of threads to 24 and the number of chunks to 48 iterations. We fixed these parameters so as they would be consistent to [8].

In the second and third experiments, we aim at analyzing the scaling capabilities of the SRR loop scheduling strategy. In the former experiment, we perform weak scaling tests to analyze how SRR scales when the chunk increases in a constant ratio of 2× with the number of threads. In the latter experiment, we carry out strong scaling tests to study how SRR performs for a fixed number of chunks. In both experiments we varied the number of threads from 2 to 24 while fixing the PDF, Kurtosis, Shuffling and Kernel performance factors.

Overall, in these three experiments, we carry out five replications of each experiment to account for the inherent variance of the measures in the experimental environment. For each replicate, the actual order in which individual runs of experiments is executed is randomly determined. This approach ensures that observations and experimental errors are independent and identically distributed (i.i.d.) random variables.

5 Experimental Results

In this section, we unveil the results of our performance analysis experiment, and then we discuss about the scaling capabilities that we observed for SRR in the weak and strong scaling experiments. Our evaluation methodology is publicly available, so all results can be easily reproduced[1]. All results presented in the paper were run on a SMP machine powered by four six-core Intel Xeon E5 processors (24 physical cores in total) with 64 GB of RAM.

[1]Experimental results are available at https://dx.doi.org/10.6084/m9.figshare.3753024.

Table 1: ANOVA of the considered performance factors.

Factor	**MSQ**	**Pr(>F)**	**Factor**	**MSQ**	**Pr(>F)**
Kurtosis	2511.12	<2.2e-16	Strategy:Kernel	52998.63	<2.2e-16
Shuffling	3094.83	<2.2e-16	PDF:Shuffling	95.04	1.121e-06
PDF	71879.79	<2.2e-16	PDF:Shuffling	95.04	1.121e-06
Strategy	118490.53	<2.2e-16	PDF:Kernel	33904.94	<2.2e-16
Kernel	2888428.00	<2.2e-16	PDF:Kurtosis	1694.24	<2.2e-16
Strategy:Kurtosis	48.90	0.002235	Kurtosis:Shuffling	12.05	0.8781
Strategy:PDF	6067.14	<2.2e-16	Kurtosis:Kernel	1201.45	<2.2e-16
Strategy:Shuffling	7329.35	<2.2e-16	Shuffling:Kernel	1649.73	<2.2e-16

5.1 Performance Analysis Overview

To guide us on the performance analysis of the SRR loop scheduling strategy, we adopted the Fisher's Analysis of Variance (ANOVA) method. With this approach, we focus on obtaining statistically significant and reproducible conclusions about the impact of the considered performance factors independently and their interactions on the response variables.

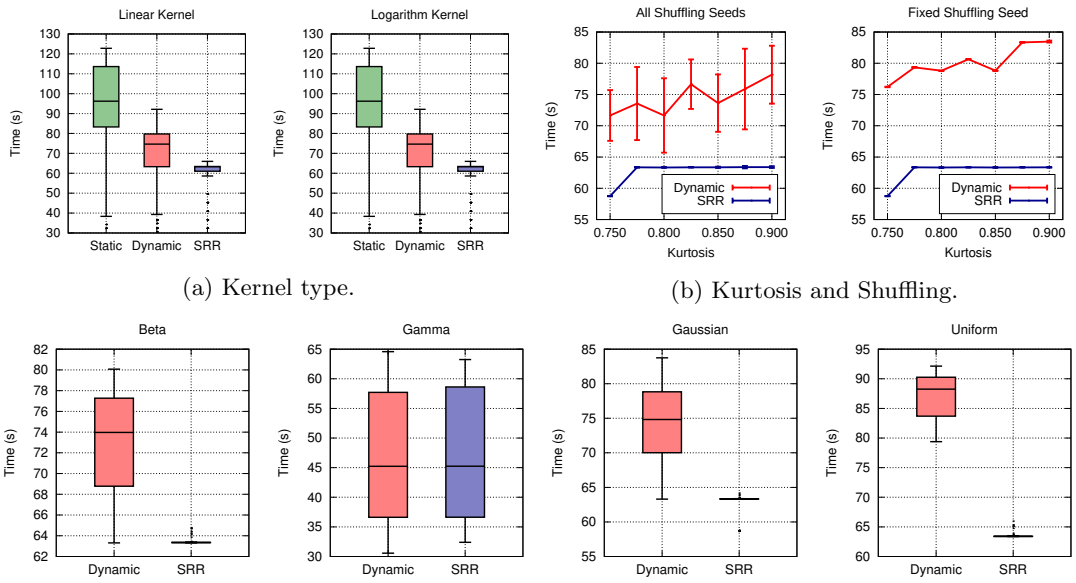
Table 1 presents the outcome of the method in the form of an ANOVA table. Results indicate that all performance factors and interactions other than *Kurtosis:Shuffling* have a highly significant impact on the time-to-solution, considering a significance level of 5% ($\alpha = 0.05$). In terms of main effects, we observed that not only they significantly impact the time-to-solution, but also their impact on the response variable is greater than their two-way interactions, accounting for 96.7% of the total performance variability.

Moreover, by taking into account the Mean Square (MSQ) of each main factor, we concluded that the application's kernel complexity is the most impacting factor on the application's time-to-solution, followed by the scheduling strategy and the performance factors related to the input workload (*i.e.,* PDF, Shuffling and Kurtosis). On the other hand, when analyzing two-way interactions between scheduling strategies and other main factors, we observed that Shuffling becomes slightly more impactful on time-to-solution than the PDF of the input workload. It is noteworthy that the variance among experiment replicates are very small, with a Mean Square of the Error (MSE) of 3261. In the following sections we carry out a top-down performance analysis of the SRR loop scheduling strategy, starting from the most impactful performance factors towards the least impactful ones.

5.2 Kernel Analysis

Figure 2a presents an overview of execution times for scheduling strategies per kernel type. These plots illustrate the variance of the time-to-solution for each kernel and strategy, varying all other factors (*i.e.,* PDF, Kurtosis and Shuffling).

When comparing the three scheduling strategies, we observed that all of them showed a similar performance behavior regardless of the kernel type. In contrast to the other strategies, the SRR scheduler presents a considerably smaller variance in execution times. Moreover, when considering the interquartile range (IQR), smaller execution times are observed with SRR, Dynamic, and Static strategies, in this order. However, when taking into account the overlap for whiskers and IQR in the boxplots, conclusions about the actual differences in execution time are misleading. Therefore, to obtain statistically significant mean differences between the strategies, we employed Tukey's method with the response variables time-to-solution and speedup, using a significance of 5%.



(a) Kernel type.

(b) Kurtosis and Shuffling.

(c) PDF type.

Figure 2: Breakdown of execution times per (a) Kernel, (b) Kurtosis/Shuffling and (c) PDF.

Indeed, results from this method unveiled that when contrasting SRR with the Dynamic and Static strategies, time-to-solution is reduced by the former strategy in 2.90s and 6.83s for the linear kernel, and in 11.19s and 33.69s for the logarithmic kernel. In addition, in means of speedup, and when compared to the Dynamic strategy, we observed that SRR performs 1.24× and 1.19× better, in the linear and logarithmic kernels, respectively. The rationale behind the overlap between IQR is discussed in Section 5.3. As a side remark, since the Static scheduler presented a worse performance than both, the Dynamic and SRR strategies, in all cases, from this point on we will carry out the analysis only with the latter two.

Finally, when contrasting one kernel type with another, we noted that there exists a significant difference between the two kernel types, regardless of the scheduling strategy used. Changing from a linear to a logarithmic kernel when using either the Static, Dynamic or SRR strategy increases the mean execution time in 73.79s, 55.21s and 46.92s, respectively.

5.3 Workload PDF Analysis

Figure 2c presents the breakdown of execution times for each strategy per PDF type. These plots refer to results with the *logarithmic* kernel when varying all levels of the other factors (*i.e.,* Kurtosis and Shuffling). Nevertheless, the following discussion applies to the *linear* kernel.

When analyzing the impact of the PDF in the performance of each strategy, we observed that for the Beta, Gaussian and Uniform PDFs, SRR presents smaller time-to-solution than the Dynamic scheduler. The greatest difference is for the Uniform PDF, where SRR has run 15.66s faster (1.41× speedup); and the smaller difference is for the Beta PDF, where SRR has run 7.56s faster (1.18× speedup). However, for the Gamma PDF, the performance of Dynamic and SRR strategies was roughly the same, with the former presenting a slightly greater variance in execution times. Indeed, in some scenarios with the Gamma PDF, SRR has presented a slower

run time than Dynamic. In the worst case, SRR suffered a performance degradation of 8.7%.

In addition, it is important to point out that the Gamma PDF is the rationale behind our inconclusive analysis in Section 5.2, when we studied the impact of the application's kernel in the performance of each strategy in means of IQR overlapping. Results with this PDF were hiding performance gains of SRR over the Dynamic in the other PDFs. Nevertheless, in this point, we observed that the Gamma PDF generates a highly irregular workload, and thus suggesting that the SRR strategy may not perform so well on asymmetric workloads.

Furthermore, when using Tukey's method to evaluate the impact of the each PDF in the performance of each strategy, we observed that for the Dynamic strategy, changing the input workload of an application from one to another greatly impacts on its execution time, regardless of the workload PDF used. On the other hand, for the SRR strategy, this difference is significant only when switching from/to a Gamma-generated input workload. Putting it differently, the time-to-solution achieved by SRR is not affected by changes in the application's input workload.

5.4 Kurtosis and Shuffling Analysis

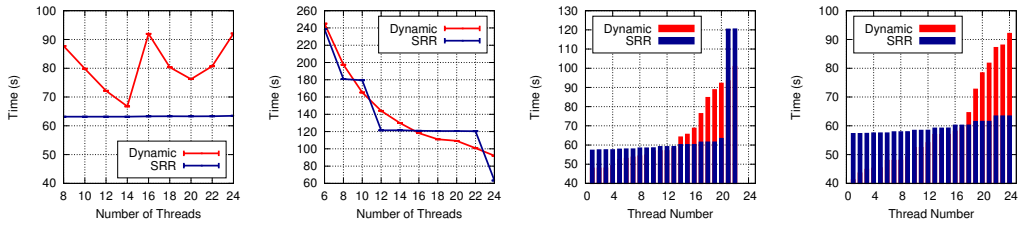
Results for the impact of Kurtosis and Shuffling on the time-to-solution are presented in Figure 2b. For both plots, the kernel type was fixed to *logarithmic* and the PDF to *Gaussian*. On the left-hand side, the time-to-solution is presented for all kurtosis, averaged by the shuffling seed value and replicates. Vertical bars refer to the standard deviation of the mean time. On the other hand, to isolate the effect of the kurtosis in the response variable, the figure on the right-hand side presents results for shuffling seed fixed to 307. The following conclusions equally apply to all other scenarios, except for those involving Gamma.

We observed that both strategies suffer from performance degradation when the kurtosis of the PDF increases. However, in all cases, this impact is greater on the Dynamic than on SRR strategy. If we consider all other scenarios, this performance degradation results in higher execution times ranging from 7.84s to 9.67s for the Dynamic strategy, on average, according to Tukey's method with significance of 5%. In means of speedup, we observed that SRR overpasses the Dynamic scheduler in performance in up to 1.5× (logarithmic kernel, Uniform PDF, 0.825 kurtosis and shuffling seed 307). When varying the seed value for iteration shuffling, we noted that the Dynamic strategy presents a greater variance on execution times and a worse performance than SRR, regardless of the seed value. The performance of Dynamic is greatly influenced by this factor due to its dynamic behavior, whereas the SRR scheduler, inherently avoids this because it sorts loop iterations beforehand.

5.5 Scaling Analysis

In this section we carry out a discussion about the scaling capabilities of the SRR strategy in the weak and strong scaling experiments. The following conclusions are based on the results that we observed for the scenario in which the SRR scheduler has performed the best (logarithmic kernel, Uniform PDF, 0.825 kurtosis and shuffling seed 307). Therefore, they reflect an upper bound scaling analysis over all the scenarios.

In the weak scaling experiment (Figure 3a), we noted small execution times when running with less than 8 threads and we did not identified any significant difference between the Dynamic and SRR scheduling strategies. However, when running with 8 threads and more, we observed that SRR delivers a constant performance when the number of threads and chunks of loop iterations increase proportionally. This result thus unveils that SRR may achieve linear weak scaling for properly large input workloads.



(a) Weak scaling. (b) Strong scaling. (c) 22 thread run. (d) 24 thread run.

Figure 3: Results for weak and strong scaling experiments.

Results for the strong scaling experiment are presented in Figure 3b (results for scenarios with less than 6 threads were omitted to improve visibility). Overall, we observed that the scaling capability of SRR is roughly similar to the Dynamic strategy, *i.e.,* time-to-solution decreases quasi-exponentially as we increase the number of threads. Nevertheless, SRR has presented steps on the graph. When the number of threads is a divisor of the number of chunks of loop iterations (48 chunks), there is a considerable drop in the execution time. However, when this number is not a divisor of the number of chunks, the execution time remains constant.

To find out more details about this peculiar behavior, we thus selected the case with 22 and 24 threads to study. However, it is important to point out that we drew the same conclusion for the other similar cases (*i.e.,* 6 and 8 threads, and 10 and 12 threads). Figures 3c and Figure 3d present the workload assigned to each thread when running with 22 and 24 threads, respectively. As it can be noted, in the first scenario, SRR ends up overloading two threads with computation, whereas in the second scenario, the input workload is evenly distributed. The rationale behind this behavior, however, comes from the SRR strategy itself. When the number of threads is not a divisor of the number of chunks of loop iterations, the very first threads that are considered in the scheduling end up with an extra pair of loop iterations, and thus load imbalance arises. Indeed, we believe that if SRR also considered the total workload assigned to each thread when scheduling loop iterations, it might had stepped out from this corner case and delivered a better strong scaling capability.

6 Conclusions and Future Work

With the ever-increasing number of loop scheduling strategies, it has become a non-trivial task to actually select the best one for a particular parallel application. Nevertheless, this challenge becomes easier to be tackled when existing strategies are extensively evaluated. Therefore, in this paper, we presented a performance and scalability evaluation of the recently-proposed workload-aware loop scheduling strategy named SRR. This strategy showed promising results for irregular applications, but had not been thoroughly assessed before. To deliver a comprehensive analysis, we coupled the synthetic kernel benchmarking technique with several rigorous statistical tools, and considered OpenMP's Static and Dynamic loop schedulers as our baselines.

We studied the impact of several factors in the time-to solution and speedup of each strategy in 672 distinct scenarios. Our experiments unveiled that SRR performs better on symmetric workloads and may achieve up to 1.9× and 1.5× better performance than OpenMP's Static and Dynamic strategies, respectively. Moreover, our results pointed out that the time-to-solution achieved by SRR is not significantly affected by changes in application's input workload, and that this strategy may deliver linear weak scaling capability for properly large input workloads.

Nevertheless, we noted that SRR performs similar to the Dynamic strategy for asymmetric workloads. Finally, when SRR faces a strong scaling scenario, it is overpassed in performance by the Dynamic scheduler, when the number of threads is not a divisor of the number of chunks of loop iterations in the irregular application.

As future works, we intend assess memory affinity and energy efficiency of the SRR strategy in NUMA and hetherogeneous platforms. Furthermore, we intend derive formal proofs on the performance bounds of this strategy. Finally, we intend to propose an enhancement in the SRR scheduler so that it can step out from the corner scenarios in which we observed that it may not perform as expected.

Acknowledgements

We would like to thank CAPES, FAPEMIG, FAPERGS and INRIA under the ExaSE project grant APQ-03206-13, CNPq under the projects grants 458530/2014-0 and 233223/2014-2, and STIC-AmSud/CAPES cooperation under EnergySFE project grant 99999.007556/2015-02.

References

[1] M. Balasubramaniam, N. Sukhija, F.M. Ciorba, I. Banicescu, and S. Srivastava. Towards the scalability of dynamic loop scheduling techniques via discrete event simulation. In *Int. Parallel and Distributed Processing Symp. Workshops*, pages 1343–1351, May 2012.

[2] L. Dagum and R. Menon. Openmp: An industry standard api for shared-memory programming. *Computational Science Engineering*, 5(1):46–55, 1998.

[3] W. Ding, Y Zhang, M. Kandemir, J. Srinivas, and P. Yedlapalli. Locality-aware mapping and scheduling for multicores. In *Int. Symp. on Code Generation and Opt.*, pages 1–12, Febi 2013.

[4] Marie Durand, François Broquedis, Thierry Gautier, and Bruno Raffin. An efficient openmp loop scheduler for irregular applications on large-scale numa machines. In *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *LNCS*, pages 141–155. 2013.

[5] A. Hajieskandar and S. Lotfi. Parallel loop scheduling using an evolutionary algorithm. In *Int. Conf. on Advanced Computer Theory and Engineering*, volume 1, pages 314–319, 2010.

[6] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. How much parallelism is there in irregular applications? In *Proc. of the SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 3–14, 2009.

[7] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. Openmp task scheduling strategies for multicore numa systems. *Int. J. High Perform. Comput. Appl.*, 26:110–124, 2012.

[8] Pedro H. Penna, Márcio Castro, Henrique Freitas, François Broquedis, and Jean-François Méhaut. Design Methodology for Workload-Aware Loop Scheduling Strategies Based on Genetic Algorithm and Simulation. *Concurrency and Computation: Practice and Experience*, 2016.

[9] Steven S. Skiena. *The Algorithm Design Manual*. 2nd edition, 2008.

[10] S. Srivastava, B. Malone, N. Sukhija, I. Banicescu, and F.M. Ciorba. Predicting the flexibility of dynamic loop scheduling using an artificial neural network. In *Int. Symp. on Parallel and Distributed Comp.*, pages 3–10, 2013.

[11] S. Srivastava, N. Sukhija, I. Banicescu, and F.M. Ciorba. Analyzing the robustness of dynamic loop scheduling for heterogeneous comp. systems. In *Int. Symp. on Parallel and Distributed Comp.*, pages 156–163, 2012.

[12] N. Sukhija, B. Malone, S. Srivastava, I. Banicescu, and F.M. Ciorba. Portfolio-based selection of robust dynamic loop scheduling algorithms using machine learning. In *Int. Parallel Distributed Processing Symp. Workshops*, pages 1638–1647, 2014.

[13] Peter Thoman, Herbert Jordan, Simone Pellegrini, and Thomas Fahringer. Automatic openmp loop scheduling: A combined compiler and runtime approach. In *OpenMP in a Heterogeneous World*, volume 7312 of *LNCS*, pages 88–101. 2012.